Plan

- Introduction & définitions
- **Mécanismes de base**
- **Notions complémentaires**

Verrou à exclusion mutuelle (Mutex)

- **Verrou simple à attente passive**
- **Primitives (pseudo-code)**
  - `void mutex_init(mutex_t *m)`
  - `void mutex_lock(mutex_t *m)`
    - Demande bloquante
  - `int mutex_trylock(mutex_t *m)`
    - Demande non bloquante
  - `void mutex_unlock(mutex_t *m)`
- **Variantes**
  - Vérification des erreurs (tentative de libérer un verrou déjà libre ou non détenu par le même processus ...)
  - Détection d'interblocage
  - Réentrance (gestion correcte de la ré-acquisition du même verrou au sein d'une section critique)

Cas d'étude :
gestion de parking N places (1/4)

```
int N = ... // nombre de places max

void entrer_parking() {
    while (N==0); // attendre
    N--;
}

void sortir_parking() {
    N++;
}
```

Code non synchronisé et donc FAUX

Cas d'étude : gestion de parking N places (2/4)

Synchronisation avec mutex

```
int N = ... // nombre de places max
mutex_t m; // à initialiser avec mutex_init
...

void entrer_parking() {
    mutex_lock(&m);
    while (N==0); // attendre
    N--;
    mutex_unlock(&m);
}

void sortir_parking() {
    mutex_lock(&m);
    N++;
    mutex_unlock(&m);
}
```

Incorrect : interblocage (monopolisation du verrou)

Cas d'étude : gestion de parking N places (3/4)

Synchronisation avec mutex

```
int N = ... // nombre de places max
mutex_t m; // à initialiser avec mutex_init
...

void entrer_parking() {
    mutex_lock(&m);
    while (N==0) {
        mutex_unlock(&m);
        mutex_lock(&m);
    }
    N--;
    mutex_unlock(&m);
}
```

```
void sortir_parking() {
    mutex_lock(&m);
    N++;
    mutex_unlock(&m);
}
```

Correct
(mais fastidieux et potentiellement inefficace)

Cas d'étude : gestion de parking N places (4/4)

- **Conclusion**
  - Les primitives de verrouillage simples ne sont pas adaptées pour la programmation aisée et efficace de tous les schémas de synchronisation
  - En particulier, on voit qu'il existe des besoins distincts :
    - Assurer que les données sont manipulées en exclusion mutuelle pour préserver leur cohérence
    - Bloquer/débloquer des processus en fonction de certaines conditions applicatives

Sémaphores (1/2)

- **Un autre mécanisme de synchronisation**
- **Un sémaphore correspond à un compteur (entier) synchronisé et une file d'attente**
- **Interface**
  - Initialisation du compteur **`count`** avec la valeur souhaitée
  - Méthode **`P`**
    - Autres noms : **`wait`**, **`sem_wait`**, **`down`** ...
    - Décrémente **`count`** et bloque l'appelant si **`count`** négatif
  - Méthode **`V`**
    - Autres noms : **`signal`**, **`sem_signal`**, **`up`** ...
    - Incrémente **`count`** et, si **`count`** négatif ou nul, réveille l'un des processus bloqués
    - Jamais bloquante
  - **Important** : Ces méthodes sont exécutées en exclusion mutuelle

Sémaphores (2/2)

```
typedef struct {
    int count;
    struct process *list; // list of blocked processes
} semaphore;
```

```
void sem_init(semaphore *s, int val) {
    s->count = val;
    s->list = NULL;
}
```

```
void P(semaphore *s) {
    s->count--;
    if (s->count < 0) {
        put(myself(), s->list);
        suspend();
    }
}
```

```
void V(semaphore *s) {
    s->count++;
    if (s->count <= 0) {
        struct process *p;
        p = get(s->list);
        wakeup(p);
    }
}
```

- **Rappel : méthodes exécutées en exclusion mutuelle**
- **Analogie avec un portillon à jetons**
  - On peut créditer des jetons à l'avance

Cas d'étude : gestion de parking N places avec sémaphore

```
int N = ... // nombre de places max
semaphore s;
...
sem_init(&s, N);
...

void entrer_parking() {
    P(&s);
}

void sortir_parking() {
    V(&s);
}
```

Cas d'étude : tampon producteur consommateur avec sémaphores (1/4)

- **On reprend l'exemple vu en introduction**
- **Tampon borné avec `BUFFER_SIZE` cases**

- **Processus producteur**
  - Peut déposer une donnée dans le tampon si le tampon n'est pas plein
- **Processus consommateur**
  - Peut récupérer une donnée dans le tampon si le tampon n'est pas vide
- **Tampon**
  - Tampon plein : 0 cases libres
  - Tapon vide : **`BUFFER_SIZE`** cases libres

Cas d'étude : tampon producteur consommateur avec sémaphores (2/4)

```
semaphore full; // supervision des cases pleines
semaphore empty; // supervision des cases vides
semaphore mutex; // exclusion mutuelle


// initialisations
sem_init(&full, 0);
sem_init(&empty, BUFFER_SIZE);
sem_init(&mutex,  1);
```

Cas d'étude : tampon producteur consommateur avec sémaphores (3/4)

Producteur

```
while (true) {
  // produce an item in
  // nextProduced

  P(&mutex);
  P(&empty);

  buffer[in] = nextProduced;
  in = (in + 1)%BUFFER_SIZE;
  count++;

  V(&mutex);
  V(&full);
}
```

Consommateur

```
while (true) {
  P(&full);
  P(&mutex);

  nextConsumed = buffer[out];
  out = (out + 1)%BUFFER_SIZE;
  count--;

  V(&mutex);
  V(&empty);

  // consume the item in
  // nextConsumed
}
```

Code INCORRECT !

Exercice : Trouver une séquence qui illustre le problème ...

Cas d'étude : tampon producteur consommateur avec sémaphores (4/4)

Producteur

```
while (true) {
  // produce an item in
  // nextProduced

  P(&empty);
  P(&mutex);

  buffer[in] = nextProduced;
  in = (in + 1)%BUFFER_SIZE;
  count++;

  V(&mutex);
  V(&full);
}
```

Consommateur

```
while (true) {
  P(&full);
  P(&mutex);

  nextConsumed = buffer[out];
  out = (out + 1)%BUFFER_SIZE;
  count--;

  V(&mutex);
  V(&empty);

  // consume the item in
  // nextConsumed
}
```

Sémaphores et verrous mutex

- **Attention :** À partir de l'exemple precedent, il ne faut pas conclure de manière générale qu'un semaphore initialisé à 1 est strictement identique à un verrou mutex.
- **Pour implementer une section critique simple, il est souvent preferable d'utiliser un verrou mutex plutôt qu'un semaphore initialisé à 1,**
- En effet, l'utilization d'un verrou offre davantage de protection contre certains bogues, notamment pour les raisons suivantes :
  - De nombreuses implementations de verrous sont réentrantes.
  - Certaines implementations de verrous permettent de détecter les tentatives de libérer non détenu par le thread appellant, ainsi que certains riques d'interblocages.